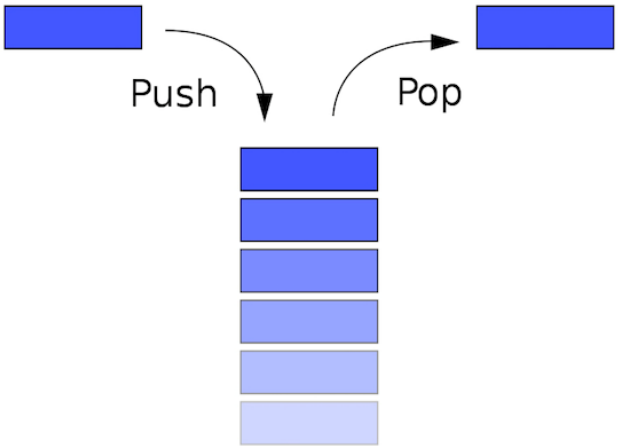
STACKS

OVERVIEW

OVERVIEW

- **What is a stack?**

Stack of dishes



Stack data structure

OVERVIEW

- **A stack data structure only allows you to insert or remove one data value at a time from the "top" of the stack**

- **Think of a pile of dishes in your cupboard**
  - We normally add or remove dishes one at a time
  - When we want to use a dish we take the top dish
  - We put clean dishes away one at a time on top of a pile

- **This pattern of data usage has two names:**
  - FILO - first in, last out
  - LIFO – last in, first out

OVERVIEW

- **A wide range of programming problems can be solved using a stack data structure**
  - We can use the stack as as a type of “memory” that records and processes patterns in user input
  - We can also use stack to store numerical data while evaluating arithmetic expressions
  - Finally, we simulate the execution of recursive functions by storing a function’s parameter values on a stack

- **Stacks can be implemented using fixed length arrays or using linked lists**
  - Arrays are faster, but linked lists can never become full

STACKS

STACK INTERFACE

STACK INTERFACE

- **The stack ADT has the following operations:**
  - Create – Initialize stack data structure
  - Destroy – Delete stack data structure
  - Push – Insert data onto the top of the stack
  - Pop – Remove the top value from the stack
  - Top – Retrieve the top value without removing it
  - IsFull – Check if the stack is at max capacity
  - IsEmpty – Check if the stack is has no data

- **The type of data stored in the stack varies by application**
  - Character – string processing
  - Float – numerical calculations

STACKS

STACK IMPLEMENTATION

ARRAY BASED

- **We create an empty stack using an array with size = 10 and a variable top = -1 which is the index of the top item**

-	-	-	-	-	-	-	-	-	-
0	1	2	3	4	5	6	7	8	9

top = -1

- **When we push a value 3 on the stack, we increment top and store the data at array[top]**

3	-	-	-	-	-	-	-	-	-
0	1	2	3	4	5	6	7	8	9

top = 0

ARRAY BASED

- **As we push more data into the stack, the array fills in from left to right and the value of top increases**

3	**1**	**4**	**1**	**-**	**-**	**-**	**-**	**-**	**-**
0	1	2	3	4	5	6	7	8	9

push 1, top = 3

3	**1**	**4**	**1**	**5**	**-**	**-**	**-**	**-**	**-**
0	1	2	3	4	5	6	7	8	9

push 5, top = 4

3	**1**	**4**	**1**	**5**	**9**	**-**	**-**	**-**	**-**
0	1	2	3	4	5	6	7	8	9

push 9, top = 5

ARRAY BASED

- **When we pop a data value off the stack, we remove the top value from the array and decrement top by one**

3	**1**	**4**	**1**	**5**	**9**	**-**	**-**	**-**	**-**
0	1	2	3	4	5	6	7	8	9

top = 5

3	**1**	**4**	**1**	**5**	**-**	**-**	**-**	**-**	**-**
0	1	2	3	4	5	6	7	8	9

pop 9 off, top = 4

3	**1**	**4**	**1**	**-**	**-**	**-**	**-**	**-**	**-**
0	1	2	3	4	5	6	7	8	9

pop 5 off, top = 3

ARRAY BASED

- **A stack is full when top = size-1**

3	1	4	1	5	9	2	6	5	3
0	1	2	3	4	5	6	7	8	9

- **A stack is empty when top = -1**

-	-	-	-	-	-	-	-	-	-
0	1	2	3	4	5	6	7	8	9

ARRAY BASED

```
class Stack
{
 public:
   // Constructors
   Stack();
   Stack(const Stack & stack);
   ~Stack();

   // Basic methods
   void Push(int Number);
   int Pop();
   int Top();
```

ARRAY BASED

```
...
   // Other methods
   bool IsFull();
   bool IsEmpty();
   void Print();

private:
   static const MAX_SIZE = 100;
   int data[MAX_SIZE];
   int top;
};
```

We declare a fixed size array here for the stack

ARRAY BASED

```
// Constructor function
Stack::Stack()
{
   for (int index=0; index<MAX_SIZE; index++)
      data[index] = 0;
   top = -1;
}
```

ARRAY BASED

```
// Copy constructor
Stack::Stack(const Stack & stack)
{
   for (int index=0; index<MAX_SIZE; index++)
      data[index] = stack.data[index];
   top = stack.top;
}
```

ARRAY BASED

```
// Destructor function
Stack::Stack()
{
   // Empty
}
```

ARRAY BASED

```
// Push method
void Stack::Push(int Number)
{
   // Check for full stack
   if (IsFull())
      return;

   // Save data in stack
   cout << "push " << Number << endl;
   data[++top] = Number;
}
```

This method ignores push if the stack is already full

This increments top before using its value to access array

ARRAY BASED

```
// Pop method
int Stack::Pop()
{
   // Check for empty stack
   if (IsEmpty())
      return 0;

   // Remove top value from stack
   cout << "pop " << data[top] << endl;
   return (data[top--]);
}
```

This method returns 0 if the stack is empty

This decrements top after using its value to access array

ARRAY BASED

```
// Top method
int Stack::Top()
{
   // Check for empty stack
   if (IsEmpty())
      return 0;

   // Return top value from stack
   cout << "top " << data[top] << endl;
   return (data[top]);
}
```

This method ignores top if the stack is empty

We are not changing value of top so data is not removed

ARRAY BASED

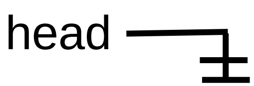
```
// True if stack is full
bool Stack::IsFull()
{
   return (top == MAX_SIZE-1);
}

// True if stack is empty
bool Stack::IsEmpty()
{
   return (top == -1);
}
```

ARRAY BASED

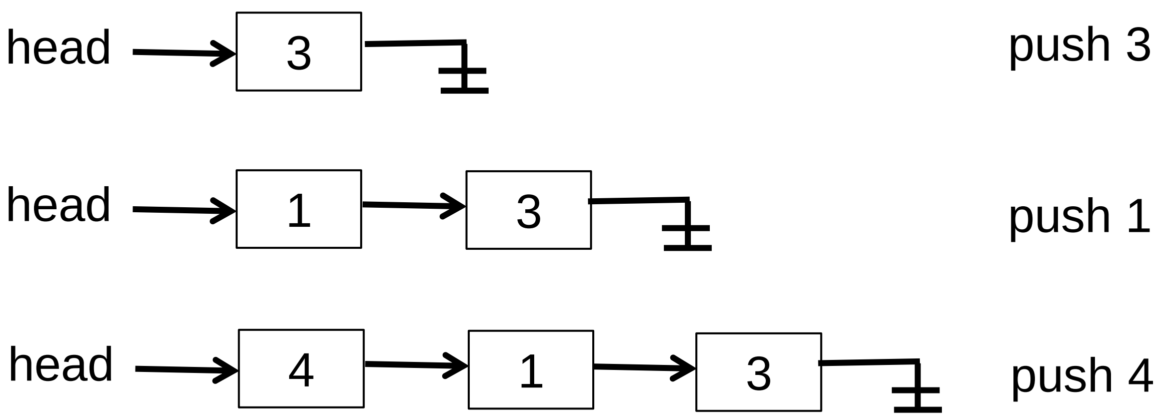
```
// Print method
void Stack::Print()
{
   cout << "stack: ";
   for (int index=0; index<=top; index++)
      cout << data[index] << ' ';
   cout << endl;
}
```

LINKED LIST BASED

- **We create an empty stack by creating an empty linked list**

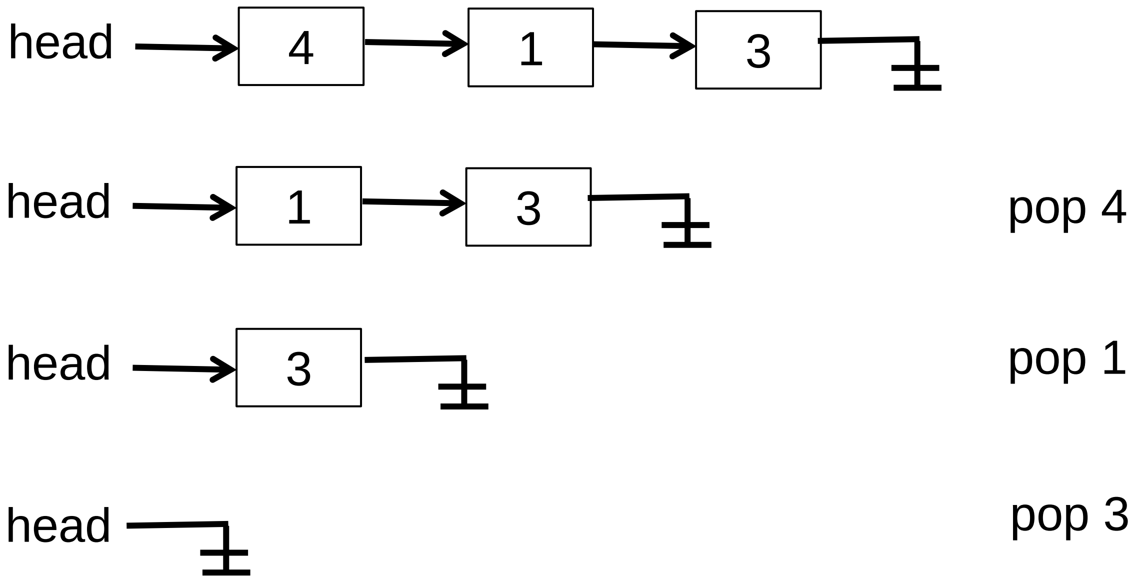


- **When we push values on the stack we insert new nodes at the head of the linked list**



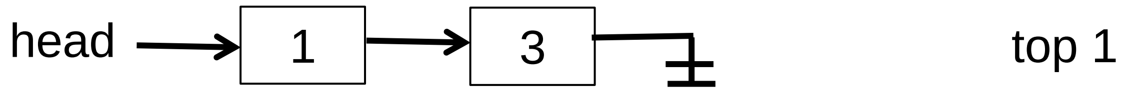
LINKED LIST BASED

- **When we pop values from the stack we delete nodes from the head of the linked list**



LINKED LIST BASED

- **To get the top of the stack, we return the first value in the linked list, without removing it from the list**



- **A linked list stack can not become full unless our program runs out of memory on the heap (hopefully never)**
- **A linked list stack is empty when the head pointer is null**

LINKED LIST BASED

```
class Stack
{
 public:
   // Constructors
   Stack();
   Stack(const Stack & stack);
   ~Stack();

   // Basic methods
   void Push(int Number);
   int Pop();
   int Top();
```

LINKED LIST BASED

```
...
   // Other methods
   bool IsFull();
   bool IsEmpty();
   void Print();

private:
   StackNode *Head;
};
```

← We only need a pointer to head of linked list

LINKED LIST BASED

```
// Node for stack data
class StackNode
{
public:
   int Number;
   StackNode *Next;
};
```

This class "breaks" the information hiding principle of OOP, but we are only going to use it in the Stack class

LINKED LIST BASED

```
// Constructor function
Stack::Stack()
{
   Head = NULL;
}
```

LINKED LIST BASED

```
// Copy constructor
Stack::Stack(const Stack & stack)
{
   // Create first node
   StackNode *copy = new StackNode();
   Head = copy;

   // Walk list to copy nodes
   StackNode *ptr = stack.Head;
```

LINKED LIST BASED

```
    while (ptr != NULL)
    {
       copy->Next = new StackNode();
       copy = copy->Next;
       copy->Number = ptr->Number;
       copy->Next = NULL;
       ptr = ptr->Next;
    }

    // Tidy first node
    copy = Head;
    Head = copy->Next;
    delete copy;
}
```

LINKED LIST BASED

```
// Destructor function
Stack::Stack()
{
   // Delete nodes from stack
   while (Head != NULL)
   {
      StackNode *Temp = Head;
      Head = Head->Next;
      delete Temp;
   }
}
```

LINKED LIST BASED

```
// Push method
void Stack::Push(int Number)
{
   // Allocate space for data
   StackNode *Temp = new StackNode;
   if (Temp == NULL) return;

   // Insert data at head of list
   Temp->Number = Number;
   Temp->Next = Head;
   Head = Temp;
}
```

This ignores push operation if we run out of memory

We insert node at the head of linked list

LINKED LIST BASED

```
// Pop method
int Stack::Pop()
{
   // Extract information from node
   if (IsEmpty()) return 0;
   int Number = Head->Number;

   // Pop item from linked list
   StackNode *Temp = Head;
   Head = Head->Next;
   delete Temp;
   return Number;
}
```

This returns 0 is stack is empty

We delete node before returning top value

LINKED LIST BASED

```
// Top method
int Stack::Top()
{
   // Extract information from node
   if (IsEmpty()) return 0;
   int Number = Head->Number;

   // Return top value without
   // removing from linked list
   return Number;
}
```

This returns 0 is stack is empty

LINKED LIST BASED

```
// True if stack is full
bool Stack::IsFull()
{
   return false;
}

// True if stack is empty
bool Stack::IsEmpty()
{
   return (Head == NULL);
}
```

LINKED LIST BASED

```
// Print method
void Stack::Print()
{
   cout << "stack: ";
   StackNode *Temp = Head;
   while (Temp != NULL)
   {
      cout << Temp->Number << " ";
      Temp = Temp->Next;
   }
   cout << endl;cout << endl;
}
```

STACKS

STACK APPLICATIONS

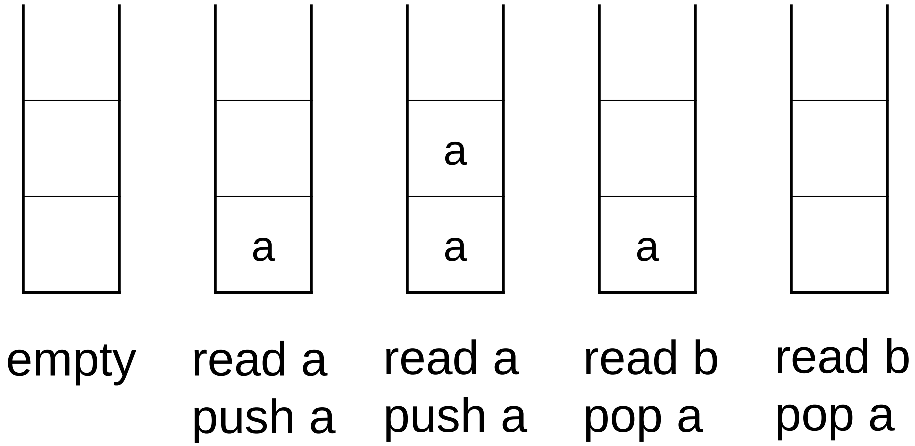
CHECKING PATTERNS

- **Goal is to see if sequence of of a's and b's is of the form:**
  - ab, aabb, aaabbb, …
  - Some number of a's followed by same number of b's
  - Pattern notation: $a^N b^N$ where N >= 1

- **Solution using stack**
  - Use stack to count the a's and b's
  - Push 'a' on the stack when you read an 'a'
  - Pop 'a' off the stack when you read a 'b'
  - Pattern matches if stack is empty at end of input

CHECKING PATTERNS

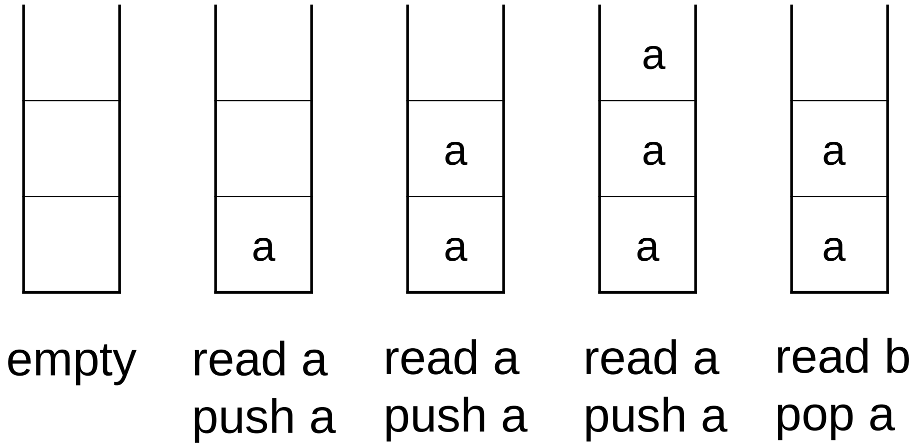
- **Example: user enters “aabb”**



- **Stack is empty at end of the so input matches the pattern**

CHECKING PATTERNS

- **Example: user enters “aaab”**



- **Stack is NOT empty so input does not match the pattern**

CHECKING PATTERNS

```
bool check_pattern(string str)
{  // Create stack
   Stack stack;

   // Read and process input string
   for(int i=0; i < str.length(); i++)
   {
      if(str[i] == 'a')
         stack.Push('a');
      else if (str[i] == 'b')
         stack.Pop();
   }
```

← This is where we implement the push and pop logic to keep track of a's and b's

CHECKING PATTERNS

```
   // Check if stack is empty
   if (stack.IsEmpty())
      return true;
   else
      return false;
}
```

CHECKING PATTERNS

- **Let's test this code with some easy cases**
  - ab – match
  - aabb – match
  - ba – no match
  - aaab – no match
- **Let's test this code with some hard cases**
  - abab – what happens?
  - aaxbb – what happens?
  - aaabbbb – what happens?

CHECKING PATTERNS

```
bool check_pattern(string str)
{  // Create stack
   Stack stack;

   // Process the a’s first
   int i = 0;
   while ((i < str.length()) && (str[i] == 'a'))
   {
      if (stack.IsFull()) return false;
      stack.Push('a');
      i++;
   }
```

CHECKING PATTERNS

```
    // Process the b’s next
    while ((i < str.length()) && (str[i] == 'b'))
    {
       if (stack.IsEmpty()) return false;
       stack.Pop();
       i++;
    }

    // Check if stack is empty and all input read
    return (stack.IsEmpty() && i == str.length());
}
```

CHECKING BRACES

- **How can we check that braces '{' and '} are nested properly in a C++ program?**
  - We could count them but that does not check ordering

- **Solution using stack**
  - Push '{' on the stack when you read an '{'
  - Pop '{' off the stack when you read a '}'
  - Pattern matches if stack is empty at end of input

CHECKING BRACES

```
bool check_braces()
{   Stack s;
    char c;
    while (cin >> c)
    {
        if (c == '{')
            s.Push('}');
        else if (c == '}')
            char ch = s.Pop();
    }
    return s.IsEmpty();
}
```

Push left brace if we see it in the input

Pop left brace if we see right brace in input

Braces are balanced if stack is empty after reading all input

CHECKING BRACES

- **Some simple testing input:**

```
if (1 == 2)
{  cout << “Impossible” << endl; }

if (1+1 == 2)
{  cout << “Addition works” << endl; }
else
{  cout << “Addition fails” << endl; }
```

CHECKING BRACES

- **What happens if we enter:**

```
while (cin >> num)
}  cout << num << endl; }
```

- **What happens if we enter:**

```
if (ch == ‘}’)
{  cout << “Found right bracket” << endl; }
else if (ch == ‘{’)
{  cout << “Found left bracket” << endl; }
```

- **We need to add stack overflow and underflow checks**

CHECKING BRACES

```
bool check_braces()
{
   const char L_BRACE = '{';
   const char R_BRACE = '}';
   Stack stack;
   char ch;
```

Define character constants to avoid typing ‘{‘ and ‘}’ in code

CHECKING BRACES

```
// Read input until EOF
while (cin >> ch)
{
   // Push brace onto stack
   if (ch == L_BRACE)
   {
      if (stack.IsFull()) return false;
      stack.Push(ch);
   }
```

CHECKING BRACES

```
    // Pop brace from stack
    else if (ch == R_BRACE)
    {
       if (stack.IsEmpty()) return false;
       if (stack.Top() != L_BRACE) return false;
       ch = stack.Pop();
    }
 }

 // Check stack is empty at end
 return stack.IsEmpty();
}
```

Check matching brace before removing from stack

POSTFIX EXPRESSIONS

- **A postfix expression is written with the operators following the values**
  - 4 7 + is equivalent to 4 + 7
  - 2 3 + 5 * is equivalent to (2 + 3) * 5

- **It is easy to evaluate postfix expressions using a stack to store input values and intermediate results**
  - When we see a value, we push it on the stack
  - When we see an operator, we pop two values from stack perform the operation and push result
  - The value on the stack at the end is the final result

POSTFIX EXPRESSIONS

- **Example: Assume the user has entered 2 3 + 5 ***

Input	Stack	Action
2	2	push 2
3	2 3	push 3
+	5	pop 2, pop 3, push 5
5	5 5	push 5
*	25	pop 5, pop 5, push 25

The top of the stack contains the answer

POSTFIX EXPRESSIONS

```
float postfix()
{
   float_stack s;
   string input;
   // Loop processing user input
   while (cin >> input)
   {
      // Handle addition
      if (input == "+")
         s.push(s.pop() + s.pop());
```

POSTFIX EXPRESSIONS

```
...
      // Handle multiplication
      else if (input == "*")
         s.push(s.pop() * s.pop());

      // Handle input value
      else
         s.push(atof(input.c_str()));
   }
   return s.top();
}
```

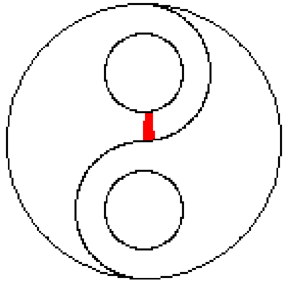
POSTFIX EXPRESSIONS

- **This solution is short and simple but it does not handle subtraction or division**
  - If user enters 6 2 – we want to calculate 6 - 2
  - s.push(s.pop() – s.pop()) is wrong
  - s.push(- s.pop() + s.pop()) is correct
  - How should we implement division?

- **Previous solution does not do error checking**
  - Should check for stack underflow in pops
  - Should check only one value on stack at end
  - See full solution on class website

STACK BASED FLOOD FILL

- **Flood fill is an algorithm used in most paint packages to fill in the interior of a line drawing**
  - User draws the object outline
  - User selects a seed point inside the object
  - User selects the desired color
  - Algorithm simulates "flooding" to fill region



[Stack based flood fill demo from Wikipedia]

STACK BASED FLOOD FILL

- **Flood fill can be implemented recursively as follows:**
  - We start at seed location (x,y) in picture
  - If pixel(x,y) is not already colored, we color this pixel and make four recursive calls to fill in adjacent locations

```
floodfill(x+1, y);
floodfill(x-1, y);
floodfill(x, y+1);
floodfill(x, y-1);
```

  - Recursion terminates if the pixel is already colored (or if the location is outside the boundary of the image)
  - If the flood fill region is large, this could result in millions of recursive calls and crash the program

STACK BASED FLOOD FILL

```
void floodfill(int picture[SIZE][SIZE],
   int x, int y, int value)
{
   // Check terminating condition
   if ((x >= 0) && (x < SIZE) &&
       (y >= 0) && (y < SIZE) &&
       (picture[y][x] != value))
   {
      // Paint this pixel
      picture[y][x] = value;
```

Checking we are inside array bounds before checking pixel

STACK BASED FLOOD FILL

```
 ...

     // Visit four neighbors
     floodfill(picture, x+1, y, value);
     floodfill(picture, x-1, y, value);
     floodfill(picture, x, y+1, value);
     floodfill(picture, x, y-1, value);
  }
}
```

Four recursive calls to visit the four adjacent locations

STACK BASED FLOOD FILL

- **Flood fill can also be implemented using a stack:**
  - We start by pushing the seed location (x,y) on stack
  - We loop until the stack is empty
  - We pop (x,y) location of current point
  - If pixel(x,y) is not already colored, we color this pixel and save adjacent locations on stack

```
push(x, y-1);
push(x, y+1);
push(x-1, y);
push(x+1, y);
```

  - We stop filling when the stack is empty
  - This method is faster and safer than recursive flood fill

STACK BASED FLOOD FILL

```
void floodfill(int picture[SIZE][SIZE],
   int startx, int starty, int value)
{
   // Push start point on stack
   Stack s;
   s.Push(startx); s.Push(starty);

   // Loop while stack not empty
   while (!s.IsEmpty())
   {
```

We push two values for (x,y) location

STACK BASED FLOOD FILL

```
// Pop next point off stack
int x = 0;
int y = 0;
s.Pop(y); s.Pop(x);

// Check if pixel is painted
if ((x >= 0) && (x < SIZE) &&
    (y >= 0) && (y < SIZE) &&
    (picture[y][x] != value))
{
```

We pop two values in reverse order to get (x,y) location

Checking we are inside array bounds before checking pixel

STACK BASED FLOOD FILL

```
            // Paint this pixel
            picture[y][x] = value;

            // Push four neighbors
            s.Push(x); s.Push(y-1);
            s.Push(x); s.Push(y+1);
            s.Push(x-1); s.Push(y);
            s.Push(x+1); s.Push(y);
        }
    }
}
```

We push two values for each of the four (x,y) locations

STACK BASED FLOOD FILL

- **We showed how flood fill can be implemented using recursion or using a stack to store pixel locations**
  - In the recursive floodfill code we visited the four adjacent (x,y) locations in RLTB order
  - In the stack based floodfill code we pushed four adjacent (x,y) locations on the stack in BTLR order but when we pop the stack we visit adjacent locations in RLTB order

- **We could reduce the stack size by checking if each (x,y) location is in bounds and colored before pushing**
  - This is a classic space-time tradeoff
  - See full solution on class website

STACKS

SUMMARY

SUMMARY

- **Stacks are a very simple abstract data type that store data in a last in first out (LIFO) order**
  - We can only store data using push
  - We can only access data using pop or top

- **Stacks can be implemented using arrays or linked lists**
  - Array implementation is much faster but can get full
  - Linked list implementation can never get full but is slower

- **Stacks can be used to solve wide range of problems**
  - Checking for symmetry, postfix evaluation, flood fill